GRAPHITE-OpenCL: Generate OpenCL Code from Parallel Loops

Alexey Kravets
ISP RAS
kayrick@ispras.ru

Alexander Monakov
ISP RAS
amonakov@ispras.ru

Andrey Belevantsev
ISP RAS
abel@ispras.ru

Abstract

OpenCL standard gains popularity, yet there is no GCC support for OpenCL code generation at the moment. This paper suggests generating OpenCL code from parallelizable loops using GRAPHITE infrastructure. When a parallelizable loop nest is found (currently, only loops without cross-iteration dependencies are handled), it is turned into an OpenCL kernel, and all necessary OpenCL calls for creating and compiling kernels and copying the needed memory to/from the device are automatically generated. The resulting binary program can be run on a multicore CPU or a GPU provided that there is an OpenCL runtime.

We provide the description of OpenCL code generation framework and heuristics used to avoid parallelizing non-profitable loops. The code that is used for OpenCL code generation from a given loop nest can be used independently from the parallelizing code. Our current experimental results on Polykernels, Polyhedron 2005, and SPEC CPU 2000 tests show some speedups, however, the overhead of running OpenCL kernels might be substantial, so the most benefit will be shown on computational programs where most of the time is spent inside parallelizable loop nests.

1 Introduction

Until recently, the major development of computer technology was to increase the efficiency of a single processor. Then, difficulties with increasing of clock speed lead to the creation of multicore architectures. OpenMP standard is successfully used for manual parallelism extraction on those using shared memory model. However, its current version is not yet suited for programming more complex heterogeneous platforms consisting of multicore CPUs and accelerators like GPUs. OpenCL (Open Computing Language) was developed to allow a programmer to write parallel code for such platforms. It is also the first open standard to provide portability and efficiency across various GPUs.

This paper introduces our work aimed at generation of OpenCL code from parallelizable loops in GCC, making it possible to automatically transform some loops to OpenCL kernels during compilation. In our point of view, this contribution is important in two ways. First, at least for some cases it allows extracting parallelism for free and thus using resources (that are often waisted otherwise) of common CPU/GPU systems. Second, its OpenCL code generation part may ease later efforts of supporting OpenCL in GCC. We describe the implementation in the following chapters, focusing on the organization of memory transfers and the cost model for determining loop nests that are profitable to parallelize. We also provide some experimental results on a multicore system and a GPU.

2 OpenCL

OpenCL programming model [4, 1, 6] allows a programmer to define special functions, called *kernels*, which can be executed in parallel on an OpenCL device. Kernels can be created, compiled and executed from a host program using the runtime API provided by the OpenCL library. Thus, a program utilizing OpenCL consists of a sequential code, which runs on a host machine, and parallel kernels that are executed on an OpenCL device. In order to achieve portability, kernels' source code is stored (or generated) in a host program and must be compiled any time this program executed. Due to this feature, a host program doesn't depend on any specific OpenCL device.

Program source code can include one or more kernel sources. A kernel can be obtained from compiled program by its name. To execute a kernel, its arguments and total number of executions must be specified. Since the host memory and the device memory may be separated, kernels can access only device buffers, which can

be created, read and written from a host machine by the corresponding OpenCL calls.

3 Graphite-OpenCL

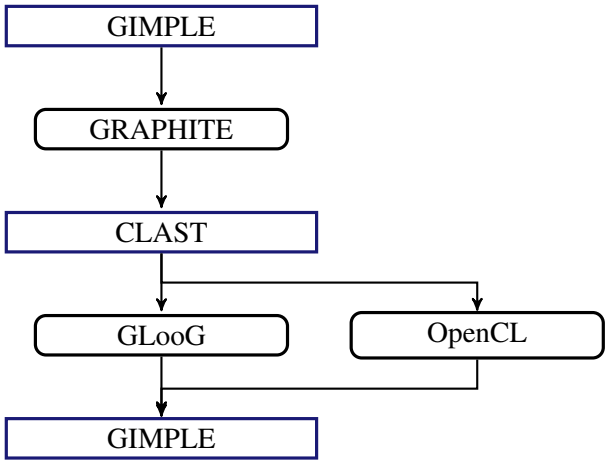

Figure 1: Code transformation scheme

Our general transformation scheme is presented in Figure 1. First, SCoPs are extracted from GIMPLE. Each SCoP represents a single entry-single exit region, keeping information about its parameters, data references and loops iteration domains. GRAPHITE transforms SCoPs, making loops more parallelizable. Its result can be obtained in the form of CLAST, a data structure that stores a SCoP as a set of loops, conditional operators and basic blocks. Some loops from this structure will be replaced by OpenCL kernels' launches, others will be restored as regular GIMPLE loops.

3.1 Selecting eligible loop nests

First of all, we should define the loop nests which will be transformed to OpenCL kernels. These loop nests must meet a number of requirements divided into correctness and performance requirements. Correctness requirements include absence of cross-iteration dependencies and host-device memory buffers synchronization. The performance requirement means that the overhead for an OpenCL kernel launch must be less than the speedup gained from its parallel execution. In order to check dependencies between loop iterations, all data references in its body are analyzed except ones to privatized variables. If there are dependencies in the loop, it must be executed on the host, but its body can contain other parallelizable loops.

3.2 Generating kernel code

After determining which nests will be transformed to kernels, we must generate correct kernel code for these nests. This problem can be divided into three subproblems:

1. Generating OpenCL code for the main program
2. Generating definitions of types and variables for the kernel code
3. Generating statements for the kernel code

All generated instructions can be divided into two classes: instructions that are generated from CLAST structures and from basic blocks. The first class includes `for` and `if` operators, the second class contains all other instructions [7, 2]. All names that are referenced in a kernel code must be defined as either local variables or kernel arguments. The first group includes

1. All variables defined inside the loop nest in the original code;
2. Loop iterators;
3. Variables generated by GIMPLE;
4. Some auxiliary variables specific for OpenCL kernels.

Consider an example of loop nest (listing 1) and the kernel source code generated from this nest (listing 2).

```
for (scat_1 = 0; scat_1 <= 99; scat_1++)
  for (scat_3 = 0; scat_3 <= 99; scat_3++)
    some_code (scat_1, scat_3);
```

Listing 1: Original loop nest

In an OpenCL kernel generated from a loop nest we need to be able to obtain values of original loops' iterators. In given example, `scat_1` and `scat_3` must be obtained from the kernel thread's global id by the `get_global_id` OpenCL call [6].

Current OpenCL implementation forbids multidimensional arrays declarations [4], so we have to generate required types definitions with pointers and the `typedef` keyword. Consider an example loop nest in listing 3 and the corresponding kernel code in listing 4.

```
__kernel void opencl_auto_function_0
      (int ocl_mod_0, int ocl_first_0,
       int ocl_mod_1, int ocl_first_1,
       int ocl_base_1, int ocl_base_0)
{
  size_t opencl_global_id = get_global_id (0);
  int scat_1 = ((opencl_global_id / opencl_base_0)
             % opencl_mod_0) * 1 + opencl_first_0;
  int scat_3 = ((opencl_global_id / opencl_base_1)
             % opencl_mod_1) * 1 + opencl_first_1;
  some_host_code (scat_1, scat_3);
}
```

Listing 2: Kernel source code

```
float A[100][100];
float B[100][100];
float C[100][100];
int foo5 ()
{
  int i,j,k;
  for (i = 0; i <= 99; i ++)
    for (j = 0; j <= 99; j ++)
      {
        float sum = 0.0;
        for (k = 0; k <= 99; k++)
          sum += A[i][k] * B[k][j];
        C[i][j] = sum;
      }
}
```

Listing 3: Sample function with multidimensional arrays

3.3 Managing OpenCL context and command queue

Valid OpenCL context and command queue must be created before any other OpenCL calls [1]. Additionally, we require that they have been created only once during program execution. Context and command queue values are stored in global variables, which can be accessed from any part of the program. At the beginning of any function with OpenCL calls the values of these variables are tested. If context is `NULL`, new context and command queue must be created and stored in appropriate variables. This code can bee seen in listing 5.

3.4 Managing memory transfers

In order to preserve semantics of the original program we must ensure that all device memory buffers accessed by an OpenCL kernel are correct before the kernel's execution, and all host memory accessed in a basic block is correct before this block's execution.

```
__kernel void
opencl_auto_function_0 (...
                  __global float *oclFTmpArg0,
                  __global float *oclFTmpArg1,
                  __global float *oclFTmpArg2)
{
  typedef __global float oclFTmpType0[100];
  oclFTmpType0 *C = (oclFTmpType0*)oclFTmpArg0;
  typedef __global float oclFTmpType1[100];
  oclFTmpType1 *A = (oclFTmpType1*)oclFTmpArg1;
  typedef __global float oclFTmpType2[100];
  oclFTmpType2 *B = (oclFTmpType2*)oclFTmpArg2;
  /*code*/
}
```

Listing 4: Types definitions for multidimensional arrays

```
cl_context __ocl_hContext = 0;

if (__ocl_hContext == 0)
{
  __ocl_hContext = clCreateContextFromType
              (0, CL_DEVICE_TYPE_GPU,
               0, 0, 0);

  clGetContextInfo (__ocl_hContext,
                CL_CONTEXT_DEVICES, 0, 0,
                &nContextDescriptorSize);

  cl_device_id * aDevices
   = malloc (nContextDescriptorSize);
  clGetContextInfo (__ocl_hContext,
                CL_CONTEXT_DEVICES,
                nContextDescriptorSize,
                aDevices, 0);

  __ocl_hCmdQueue
   = clCreateCommandQueue (__ocl_hContext,
                     aDevices[0], 0, 0);

}
```

Listing 5: Context initialization at the beginning of the function

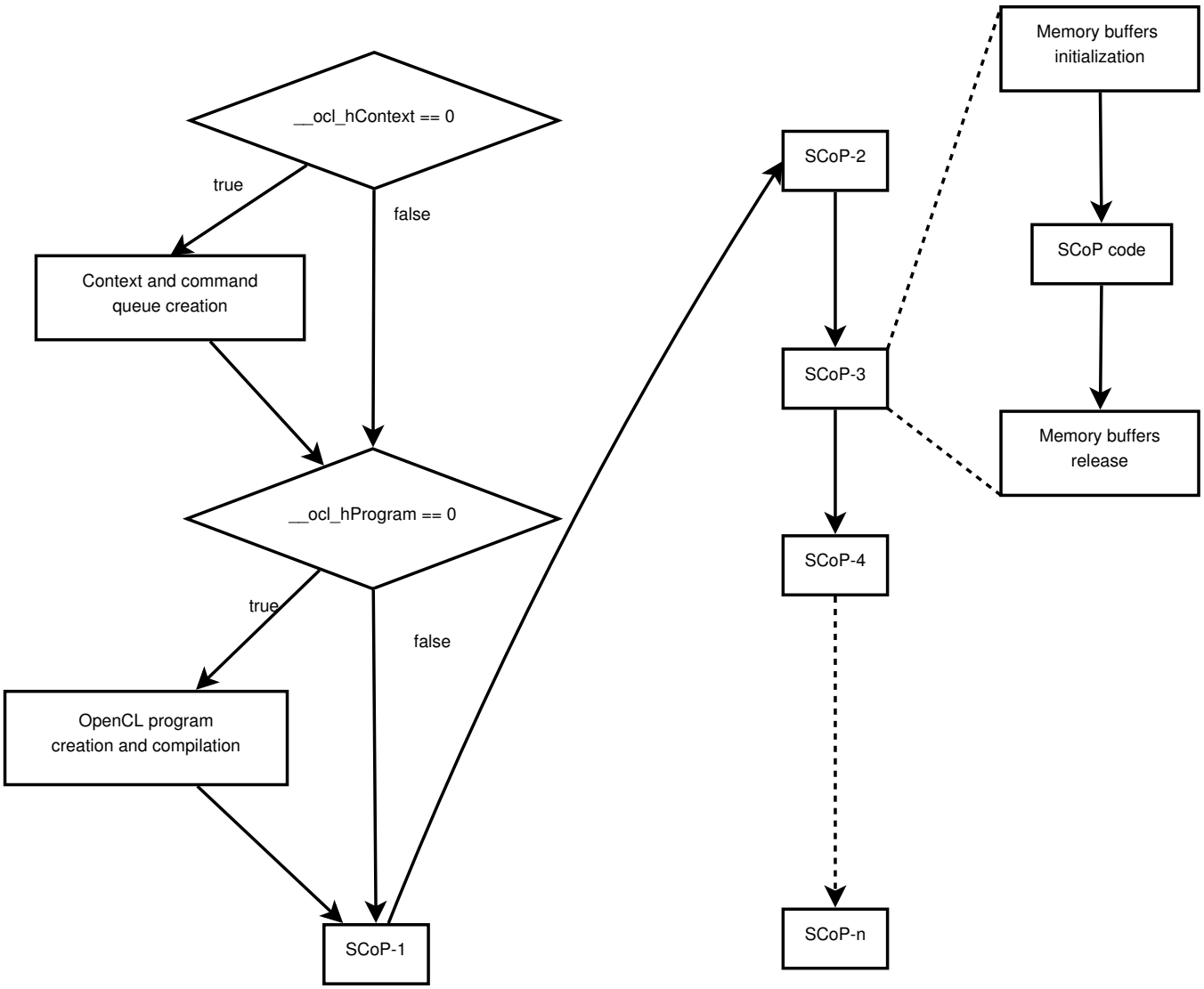

Figure 2: Structure of code generated from CLAST

A device memory buffer is correct in a given point of a program iff it contains the same data as the corresponding host memory in the original program. A host memory buffer is correct in a given point of a program iff it contains the same data as this memory object in the original program. The simplest way to meet these requirements is to copy all required buffers from the host to the device just before any kernel's execution and back from the device to the host right after the execution. The main problem with this approach is that those memory transfers can be quite expensive. So it's important to minimize their number. For this purpose we store information about *correctness* of each memory buffer (either on the host or on the device) during the SCoP analysis and use it to reduce the number of transfers.

3.4.1 Performing memory initialization

A device memory buffer must be allocated for each array or pointer used on the device in the current SCoP at the beginning of this SCoP. For some buffers their size can be calculated only at the beginning of the SCoP, so they must be allocated in each SCoP with OpenCL calls.

One of the most important problems in this part is to determine correct base object and buffer's size. The base object for an array reference is an array, while for pointer references the base object can be either an array, if it can be determined, or a pointer. Current GRAPHITE-OpenCL implementation supports only one-level pointers. The buffer size is calculated as

the maximal offset from the base object in data references plus one. All created buffers are initialized with the data from corresponding host memory.

3.4.2 Managing memory transfer during program execution

During program execution some calculations are performed on the host and some on the device. In order to get a correct result we must ensure that all required device buffers contain correct data before the kernel execution and all host memory buffers contain correct data before the host loop nest execution.

Selecting operations, that require memory transfer

The correctness information about the state of each buffer changes according to the following rules:

1. At the beginning all buffers on host and device are correct.
2. If there is a write reference in the kernel to some buffer on the device, then the corresponding buffer on the host is marked as incorrect.
3. If there is a write reference in the host program to some buffer on the host, then the corresponding buffer on the device is marked as incorrect.
4. If there is a memory copy from a host buffer to the corresponding device buffer, then this device buffer is marked as correct.
5. If there is a memory copy from a device buffer to the corresponding host buffer, then this host buffer is marked as correct.

And these rules ensure that all computations are performed on correct data:

1. Before the kernel execution all device buffers referenced in this kernel must be correct.
2. Before the host code execution all host buffers referenced in this part of code must be correct.
3. Before copying memory from a host buffer to the device this host buffer must be correct.
4. Before copying memory from a device buffer to the host this device buffer must be correct.

In order to calculate the correct placement for memory transfers, let us examine a loop nest as a tree. Each loop is represented as a tree node, and its body is divided into inner loops and basic blocks and is represented as child nodes and leaf nodes respectively.

Also, let us define three sets for each tree node B – last(B), pred(B) and modify(B), which can be calculated by the algorithm in listing 6. So, last(B) is the last leaf in subtree B (or B itself, if B is a leaf), modify(B) includes all memory buffers modified in a loop or a basic block corresponding to B, and pred(B) includes all leaf trees corresponding to basic blocks which preceed the basic block corresponding to B.

if $is\_basic\_block\_p(B)$ **then**
  last(B) = {B}
  modify(B) = {}
  for all M : $memory\_buffer$ **do**
    if $modified\_in\_block(M,B)$ **then**
      modify(B) = modify(B) ∪ M
    end if
  end for
else
  {B is a loop nest, replaced by kernel}
  last(B) = last($get\_last\_child(B)$)
  modify(B) = {}
  for all $C \in get\_children(B)$ **do**
    modify(B) = modify(C) ∪ M
  end for
end if
if $first\_child(B)$ **then**
  $P = get\_parent(B)$
  pred(B) = pred(P) ∪ last($get\_last\_child(P)$)
else
  pred(B) = last($get\_previous\_child(B)$)
end if

Listing 6: last, pred and modify calculation for a single tree node

Based on these sets we can determine which buffers must be transferred from the host to the device and back. The algorithm in listing 7 determines which buffers must be transferred before executing basic block B on the host or a kernel on the device.

$pred = \text{pred}(B)$
$result = \{\}$
for all $M : memory\_buffer$ **do**
  if $access\_in\_block(M,B)$ **then**
    for all $block \in pred$ **do**
      if $is\_incorrect\_after(M,block)$ **then**
        $result = result \cup M$
      end if
    end for
  end if
end for

Listing 7: Calculation of memory buffers to transfer

Determinating memory transfer placement
After determining which kernels or basic blocks require memory transfer we must select its placement. In order to minimize the number of such transfers during execution, each transfer is placed in the outermost loop of the given loop nest with respect to the correctness requirement. The algorithm which calculates memory transfer placement for a given node B and a memory object M is presented in listing 8

$current = B$
while $m \notin \text{modify}(get\_parent(current))$ **do**
  $current = get\_parent(current)$
end while
$\{M$ modified in loop, which contains $current\}$
return $get\_previous\_child(current)$

Listing 8: Memory transfer placement calculation

3.4.3 Cleaning memory buffers

All device memory buffers created for the given SCoP must be released at the end of this SCoP. If some host buffers aren't correct, copying from corresponding device buffers must be performed before releasing.

3.5 Managing kernels

Each compiled OpenCL program is stored in the corresponding static variable, so each OpenCL program can be compiled only once during main program execution. At the beginning of each function with OpenCL calls this variable is compared with NULL. If it is NULL, then the OpenCL program for this function will be compiled and stored in this variable, otherwise the stored value will be used. Kernels can be created from the compiled program at any time, because it's a cheap operation.

```
arg_type _ocl_scaral_arg = arg;
clSetKernelArg (kernel, index, sizeof (arg_type),
            &_ocl_scaral_arg);
```

Listing 9: Scalar argument

Scalar arguments can be passed directly to an OpenCL kernel taking into account that they must be passed through the pointers, as can be seen in the example of listing 9. For arrays and pointers, the pointer to the appropriate device buffer must be passed as a usual scalar argument.

3.6 Cost model

Consider the task of loop nest selection with the addition requirement of performance, that is, nest to kernel transformation shouldn't degrade the performance. In order to detect whether the transformation is profitable, we are using the set of heuristics.

3.6.1 Nesting ratio

Any kernel launch requires some additional operations and thus produces overhead. So, it's not reasonable to launch a lot of small kernels. Consider an example in listing 10.

```
int foo0 ()
{
  for (int i = 0; i < N; i ++)
    for (int j = 0; j < N; j ++)
      for (int k = 0; k < M; k++)
        {
          host_code ();
          for (int s1 = i; s1 < j; s1++)
            for (s2 = 0; s2 < L; s2 ++)
              use (i, j, k, s1, s2);
        }
}
```

Listing 10: Example of a loop nest which should not be replaced by a kernel because of its nesting ratio

Current test results show that in such SCoPs replacing the inner nest (the only one without dependencies) by the kernel can seriously degrade performance. Let $D_{inner}(N)$ is the depth of loop nest N and $D_{outer}(N)$ is

the depth at which this nest is located in the SCoP. Current implementation avoids transformation of nests with $D_{outer} \geq D_{inner}$.

3.6.2 Memory transfer

```
int foo1 ()
{
 host_use_a_b ();
 put_a_b_to_device ();
 for (int i = 0; i < N; i ++)
  for (int j = 0; j < N; j ++)
   a[i][j] = b [i][j];
 host_use_a_b ();
}
```

Listing 11: Example of a loop nest which should not be replaced by a kernel because of memory transfer cost

Consider an example in listing 11. This code can easily be transformed to an OpenCL kernel, but this is not reasonable from the memory transfer point of view. As we can see, there is only one reference for each element of `a` and `b`, while we have to copy `a` to the device before computation and `b` to the host after computation. In this example the overhead for required OpenCL calls is similar to the whole computation time on the host.

```
int foo2 ()
{
 host_use_a_b ();
 put_a_b_to_device ();
 for (int k = 0; k < N; k ++)
  {
   some_host_code ();
   for (int i = 0; i < N; i ++)
    for (int j = 0; j < N; j ++)
     a[i][j] += b [i][j] * k;
  }
 host_use_a_b ();
}
```

Listing 12: Example of a loop nest which should be replaced by an OpenCL kernel

In the example in listing 12 we have only one copying the from host to the device and back for N kernel launches, so the overhead is acceptable.

Another example in listing 13 shows that multiple kernel executions for a single memory transfer is not necessary. This example is a usual matrix to matrix multiplication, which can be efficiently parallelized with OpenCL.

As a conclusion we can define the set of rules to decide whether we should transform a loop nest to an OpenCL kernel. Here is the list of sufficient conditions, which must be met for all data references in the loop nest.

```
int foo3 ()
{
 host_use_a_b_c ();
 put_a_b_c_to_device ();
 for (int i = 0; i < N; i ++)
  for (int j = 0; j < N; j ++)
   {
    int red = 0;
    for (int k = 0; k < N; k ++)
     red += a[i][k] * b[k][j];
    c[i][j] = red;
   }
}
```

Listing 13: Example of a loop nest which should be replaced by an OpenCL kernel

- The memory transfer for the loop nest should be located outside the outer loop of this nest (see listing 12 for example)
- The depth of the innermost data reference in the loop nest should be greater then the data dimension (see listing 13 for example)

4 Results

The resulting compiler with GRAPHITE-OpenCL implementation has been tested on three test sets:

- PolyKernels [5];
- SPEC CPU2000 [3];
- Polyhedron 2005 Benchmark Suite [8].

4.1 CPU benchmarks

The tests were run on a quad-core processor (Intel Core 2 Quad CPU), so the maximum possible acceleration is a four-fold acceleration compared to the same program performed entirely on a single core. When the host CPU and the OpenCL device are the same, it's not necessary to transfer memory between the device and the host, and we use that to our advantage. In this situation kernels have direct access to the host memory.

4.1.1 PolyKernels

The results for this test set are given in table 1.

These tests are computational programs where the bulk of calculations is concentrated in a single loop nest. Such nests, when they don't have data dependencies, can be efficiently parallelized with OpenCL.

Title	Time		Speedup
	GRAPHITE	OpenCL	
jac.c	5.98	5.96	x1.00
jac2d.c	85.45	88.85	x0.96
adi.c	1.97	1.81	x1.08
fdtd1d.c	0.99	0.89	x1.11
fdtd-2d.c	3.70	3.71	x0.99
gs.c	1.41	1.43	x0.98
gemver.c	0.77	0.60	x1.28
lud.c	1.00	1.00	x0.99
mmm.c	22.34	7.45	x2.99
mvt.c	0.59	0.51	x1.15
sor.c	15.08	10.09	x1.49
ssymm.c	14.58	4.90	x2.97
ssyr2k.c	17.91	19.22	x0.93
ssyrk.c	64.57	64.62	x0.99
strmm.c	64.06	60.03	x1.06
strsm.c	63.20	59.91	x1.05
tmm.c	19.35	7.36	x2.62
trisolv-if.c	0.07	0.07	x0.99
trisolv.c	25.82	26.96	x0.95

Table 1: PolyKernels results on CPU

4.1.2 SPEC CPU2000

The results of this test set are given in table 2 and 3.

Results without heuristics
SPEC tests contain many simple loop nests which can be parallelized, but it's not profitable. When we are not using profitability heuristics, the overhead for replacing all possible loop nests with OpenCL kernels is too high.

As an example let us analyze one of these tests, `parser`. This test contains 6 parallelizable loops that can be replaced by kernels. These loops are located in three different functions, so a new OpenCL program must be built three times. Also, executing any of these loop nests as a kernel requires a memory transfer with the cost close to the original loop nest execution cost (like the nest in listing 11). Thus, without applying profitability heuristics the restuling overhead seriously degrades performance.

Title	Time		Speedup
	GRAPHITE	OpenCL	
wupwise	88.5	82.6	x1.07
swim	133	132	x1.00
mgrid	179	180	x0.99
applu	143	144	x0.99
mesa	58.7	58.8	x0.99
galgel	67.1	73.1	x0.91
art	41.0	40.8	x1.00
facerec	99.8	99.9	x0.99
lucas	92.3	85.5	x1.07
fma3d	145	158	x0.91
apsi	152	154	x0.98
vpr	73.8	73.9	x0.99
mcf	95.9	95.6	x1.00
parser	132	277	x0.47
eon	46.0	46.6	x0.98
gap	51.6	51.8	x0.99
vortex	79.9	80.5	x0.99
twolf	111	113	x0.98

Table 2: SPEC CPU2000 without heuristics results on CPU

Results with heuristics
With heuristics enabled, most parallelizable loop nests are discarded, as it was described above, so no OpenCL kernels can be extracted from loops nests in these tests.

4.1.3 Polyhedron 2005 Benchmark Suite

The results of this set of tests are similar to the previous ones.

4.2 GPU benchmarks

The same test sets were executed on a host machine with Intel Pentium D CPU 3.40GHz and an nVidia

Title	Time		Speedup
	GRAPHITE	OpenCL	
wupwise	81.8	83.1	x0.98
swim	131	131	x1
mgrid	181	180	x1.00
applu	145	146	x0.99
mesa	58.6	58.2	x1.00
galgel	67.6	67.4	x1.00
art	41.6	42.9	x0.96
equake	55.7	56.1	x0.99
facerec	100.0	100	x1
ammp	119	119	x1
lucas	84.3	84.5	x0.99
fma3d	147	157	x0.93
apsi	153	153	x1
gzip	97.6	99.0	x0.98
vpr	74.5	74.1	x1.00
gcc	53.8	52.8	x1.01
mcf	96.3	98.8	x0.97
crafty	38.6	39.2	x0.98
parser	131	133	x0.98
eon	45.7	46.6	x0.98
perlbmk	70.4	69.8	x1.00
gap	51.6	51.6	x1
vortex	79.9	80.9	x0.98
bzip2	76.9	76.7	x1.00
twolf	112	112	x1

Table 3: SPEC CPU2000 results with heuristics on CPU

GPU (GeForce GTX 260). In this case, memory transfers must be performed, because the host machine and the OpenCL device have different memory spaces. We are providing here only results of PolyKernels runs because profitability heuristics have prevented practically all OpenCL transformations for SPEC and Polyhedron benchmarks.

4.2.1 PolyKernels

In these results, for some tests the memory transfer cost reduces the gain from parallel execution compared with the CPU results. On the other hand, execution time for some tests has been decreased several times.

Title	Time		Speedup
	GRAPHITE	OpenCL	
ac	11.68	11.79	x0.99
aermod	38.14	38.72	x0.98
air	7.95	8.05	x0.98
capacita	51.41	50.66	x1.01
channel	2.30	2.41	x0.95
doduc	44.07	43.53	x1.01
fatigue	9.33	9.37	x0.99
gas_dyn	10.28	10.31	x0.99
induct	48.81	48.59	x1.00
linpk	23.40	24.28	x0.96
mdbx	13.76	12.96	x1.06
nf	22.22	22.22	x1
protein	42.28	41.37	x1.02
rnflow	30.66	30.61	x1.00
test_fpu	13.30	12.89	x1.03
tfft	2.46	2.43	x1.01

Table 4: Polyhedron 2005 Benchmark Suite results on CPU

Title	Time		Speedup
	GRAPHITE	OpenCL	
jac.c	10.75	9.36	x1.14
jac2d.c	86.47	86.81	x0.99
adi.c	6.84	6.83	x1.00
fdtd1d.c	1.51	1.53	x0.99
fdtd-2d.c	3.62	3.65	x0.99
gs.c	1.50	1.50	x1.00
gemver.c	1.09	1.89	x0.57
lud.c	1.27	1.27	x1.00
mmm.c	34.69	0.88	x39.00
mvt.c	2.39	1.13	x2.10
sor.c	2.02	2.02	x1.00
ssymm.c	31.74	11.86	x2.67
ssyr2k.c	15.57	15.54	x1.00
ssyrk.c	50.60	50.26	x1.00
strmm.c	55.54	55.59	x0.99
strsm.c	52.87	52.84	x1.00
tmm.c	32.57	10.83	x3.00
trisolv_if.c	0.10	0.10	x0.99
trisolv.c	42.81	42.79	x1.00

Table 5: PolyKernels results on GPU

4.3 Summary

As can be see from the above tests, overhead costs for executing a huge number of kernels can be quite high.

In certain cases it exceeds the gain from parallel execution of the transformed loops. Thus, the best result can be reached on computational programs, which primary consist of deep loops nests. For such programs the overhead time is small in comparison with the time spent on calculations.

4.4 Future work

There are several ways to improve our current implementation. First, we are going to increase the number of loops which can be replaced by kernels. This can be done either be reducing the kernel launch overhead or by handling some dependencies in the kernel body. Second, we can use OpenCL event mechanism for synchronization between memory operations and kernels executions. This can reduce overhead by removing some of the synchronization calls. Also, current implementation uses GRAPHITE branch as of 2010-06-07. Merging it with the branch requires some changes in GRAPHITE-OpenCL because some of the internal GRAPHITE data structures has been changed. We plan to complete the merge in the near future as the first step of contributing the implementation for GCC trunk.

5 Conclusion

We provided the description of OpenCL code generation framework and heuristics used to avoid parallelizing non-profitable loops. The code that is used for OpenCL code generation from a given loop nest can be used independently from the parallelizing code. Current experimental results on Polykernels, Polyhedron 2005, and SPEC CPU 2000 tests show some speedups, however, the overhead of running OpenCL kernels might be substantial, so the most benefit will be shown on computational programs where most of the time is spent inside parallelizable loop nests. Our future work is extending the implementation and contributing it to GCC trunk, the first step for this being the merge of GRAPHITE-OpenCL with the GRAPHITE branch.

References

[1] ATI. *OpenCL Programming Guide*, March 2010.

[2] C. Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, September 2004.

[3] Standard Performance Evaluation Corporation. SPEC CPU2000, 2000. `http://www.spec.org/cpu2000/`.

[4] Khronos OpenCL Working Group. *The OpenCL Specification*, aug 2008.

[5] IBM. PolyKernels. `http://groups.google.com/group/gcc-graphite/`.

[6] NVIDIA. *NVIDIA OpenCL JumpStart Guide*, April 2009.

[7] Jan Sjodin, Sebastian Pop, Harsha Jagasia, Tobias Grosser, and Antoniu Pop. Design of GRAPHITE and the Polyhedral Compilation Package. *GCCSummit'09*, 2009.

[8] Polyhedron Software. Polyhedron 2005 Benchmark Suite, 2005. `http://www.polyhedron.com/polyhedron_benchmark_suite1html`.